Theoretical Computer Science 75 (1990) 15-43
North-Holland

COMPARATIVE METRIC SEMANTICS FOR CONCURRENT PROLOG*

J.W. de BAKKER and J.N. KOK**

Centre for Mathematics and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract. This paper shows the equivalence of two semantics for a version of Concurrent Prolog with non-flat guards: an operational semantics based on a transition system and a denotational semantics which is a metric semantics (the domains are metric spaces). We do this in the following manner. First a uniform language $\mathscr{L}$ is considered, that is a language where the atomic actions have arbitrary interpretations. For this language we define an operational and a denotational semantics, and we prove that the denotational semantics is correct with respect to the operational semantics. This result relies on Banach's fixed point theorem. Techniques stemming from imperative languages are used. Then we show how to translate a Concurrent Prolog program to a program in $\mathscr{L}$ by selecting certain basic sets for $\mathscr{L}$ and then instantiating the interpretation function for the atomic actions. In this way we induce the two semantics for Concurrent Prolog and the equivalence between the two semantics.

1. Introduction

"Pure" logic programming (LP) has by now a well-established semantic theory, described in, e.g. [26, 2, 4]. Traditionally, at least two varieties of semantics are distinguished, viz. the "declarative" (minimal models, least fixed point of an immediate consequence operator) and "operational/procedural" (SLD resolution), and for pure LP, it is a standard result that these semantics all coincide. For logic *programming languages*—with the emphasis now on programming *language* rather than on the underlying mathematical framework of pure LP—the situation is much less clear. Already for PROLOG, the prime example of a *sequential* language with its prescribed execution order (left-first selection and depth-first searching) and cut operator, the development of models situated in the tradition of programming language semantics, viz. *operational* and *denotational*, and the establishment of the relationships between these models is a topic of recent and current research (e.g. [19, 14, 5, 13]).

Next we consider the field of parallel logic languages. The most well known parallel logic languages are Concurrent Prolog [35], PARLOG [7, 31] and Guarded Horn Clauses [38]. Further offsprings include (see [31] for more details and comparative language design comments) Flat Concurrent Prolog [27] and Flat Guarded

* This work was carried out in the context of ESPRIT 415: Parallel Architectures and Languages for Advanced Information Processing—a VLSI-directed approach and in the context of LPC: the Dutch National Concurrency Project, supported by the Netherlands Organization for Scientific Research (N.W.O.), grant 125-20-04.

** Current address: University of Utrecht, Department of Computer Science, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands.

Horn Clauses (discussed in [31]). Finally, we mention TFCP (Theoretical Flat Concurrent Prolog), described in [36].

Parallelism in LP languages brings along the well-known (from the field of imperative laguages) phenomena such as synchronization, suspension and deadlock, sending and receiving of messages, and process creation. Accordingly, it may be more advantageous to address the semantics issues in parallel LP following the tradition in imperative languages (emphasizing "control") rather than that of pure LP (emphasizing "logic") (cf. [8]).

For operational semantics the method of Structured Operational Semantics [18, 30] has become the standard tool. Systems of (possibly labeled) *transitions* are embedded into syntax directed deductive systems, providing a concise, powerful and flexible tool, as demonstrated by numerous applications (for parallel logic languages we mention [33]). For denotational semantics we use metric structures as our main tool. The motivation for this is, briefly, the following. In a setting with parallelism, some form of "history" of the computation (be it (sets of) sequences or traces, trees etc.) always plays a key role. Now, firstly, histories allow a natural metric (the longer the histories remain the same, the smaller their distance). Secondly, with repect to this metric many functions which play a role in our semantic domains are *contracting*. Contracting functions have *unique fixed points*, a fortunate circumstance which faciltates definitions of (the meaning of) recursion and of semantic operators, and which leads to a uniform and powerful technique in comparing concurrency semantics [22, 9].

A well-known phenomenon from imperative concurrency is that of *deadlock* (in LP returning as *failure*), inducing the need for a model which embodies more structure than just (sets of) sequences. A large variety of such "branching time" models has by now been proposed, including ready sets, failure sets, and (synchronization) trees (see [29] for a comparison). In the case where programming notions requiring branching time are combined with state transformations, the need for Plotkin's *resumptions* arises. We have developed our own (metric) way of solving domain equations which are at the bottom of such resumptions (described in [10] or [1]). The introduction of committed-choice in parallel logic languages is a cause of deadlock (see for example [16] for an analysis of this phenomenon).

In [21] we developed a denotational semantics for a version of Concurrent Prolog, employing the metric techniques (domains of processes in the resumptions style, contracting functions etc.) of [10] and successors. The branching structure built up as result of a computation before a *commit* is encountered, is collapsed, at the moment of such an encounter, into a set of streams. The paper [17] develops, for the language TFCP [36], operational and denotational semantics, the latter based on failure sets. Moreover, a fully abstractness theorem relating the two is presented. The third investigation we mention follows the approach of declarative semantics. In [25], a comprehensive analysis is provided of a number of synchronization mechanisms in parallel logic languages. This is achieved by defining a "universal" language which incorporates all the features required to model the various syn-

chronization mechanisms, and which contains as proper subsets Concurrent Prolog, Flat CP, GHC (and, a fortiori, Pure Horn Clause Logic). This language is given the usual semantic definitions on an extended Herbrand Universe, and all the standard results are shown to hold. The paper [25] is an extension of [24], dealing with the declarative semantics of logical read-only variables. Recently, declarative semantics for Flat Guarded Horn Clauses was also proposed [23].

In this paper we develop an operational and a denotational semantics for a language $\mathscr{L}$. This language is *uniform* in the sense that the elementary actions can have arbitrary interpretations. Another feature of $\mathscr{L}$ is that we have an operator that turns its argument (any, possibly complex, statement s) into an elementary action or (control) *atom*, denoted by $[s]$; hence our emphasis on *atomicity* in this investigation. We provide a proof of the correctness of the denotational semantics with respect to the operational semantics (we show that there exists a restriction operator which relates the two). The operational semantics $\mathscr{O}$ is based on a transition system. The denotational semantics $\mathscr{D}$ is a metric semantics: the domains are metric spaces. A key role is played by *contractions*; they are used in almost all definitions. We have used *uniform abstraction*; in order to obtain the two semantics for Concurrent Prolog, we interpret the abstract sets of $\mathscr{L}$. For example, the set of elementary actions B will be the set of pairs (a_1, a_2) of (logical) atoms. The intended meaning of such a pair (a_1, a_2) of atoms is that we have to unify a_1 and a_2. We then show how to translate a Concurrent Prolog program to a program in the uniform language $\mathscr{L}$. The denotational model that is induced in this way (from the denotational model for $\mathscr{L}$) resembles the model given in [21]. We also have an induced operational semantics and an induced relation between the two semantics. Figure 1 shows the relations. Note that the heavy lines in this figure refer to induced mappings only.

We think that the uniform abstraction procedure of first giving semantics to a uniform language and then the interpretation, gives more insight into the model. Moreover, we have the automatic link with an operational model.

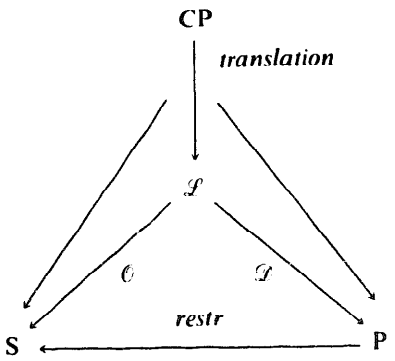

Fig. 1. Overview of the models.

The idea of a translation of Concurrent Prolog has already been presented in [6]. In that paper a translation to Milner's CCS (Calculus of Communicating Systems) is provided. The recursion structure that is used in the paper is different: a clause is modeled by an agent which tries continuously to apply itself. In our model the equivalent of a set of clauses is a set of (recursive) procedures. The model in [6] is based on synchronous communication, which is not present in our model.

We treat a larger subset of Concurrent Prolog than [17]. The main difference is that we allow non-flat guards. This leads to more complex semantic domains: we have to introduce the notion of atomicity. For semantic models of flat Guarded Horn Clauses we refer to [28]. He gives a semantics based on interactions with the outside world (a kind of assumption/commitment pairs). One of the nice points of [17] is that it makes clear what can be observed from a Concurrent Prolog program; for example that we can distinguish between failure and deadlock. They prove that their semantics is fully abstract with respect to the operational semantics. If we take the same observation criteria, we can adapt our semantic model (restricted to the subset considered by [17]) in such a way that it is fully abstract along the lines of the methods described in [32]. A point of further research is whether or not non-flat guards influence these results. Following [3] we recall that in the case of unbounded non-determinism (caused by non-flat guards) it might be impossible to assign a fully abstract semantics.

We give an outline of the rest of our paper. Metric topological preliminaries are given in Section 2. Section 3 describes the language $\mathscr{L}$ with its operational semantics $\mathscr{O}$ and in Section 4 the denotational semantics is defined. Section 5 gives the relationship between $\mathscr{O}$ and $\mathscr{D}$. Finally, Section 6 provides the translation from Concurrent Prolog to $\mathscr{L}$. There are two appendices: the first one shows the compactness of a certain set (this result is used to show that one of the sematic models is well-defined) and in the second appendix we treat the extended unification of Concurrent Prolog.

2. Metric preliminaries

We give in this section some basic definitions and properties about metric spaces. Let $\mathbb{N}$ be the set of natural numbers. For further reference we suggest [15].

Definition 2.1 (*Metric spaces*). A metric space is a pair (M, d) with M a non-empty set and d mapping $d: M \times M \to [0, 1]$ (a metric distance), which satisfies the following properties:

(1) $\forall x, y \in M[d(x, y) = 0 \Leftrightarrow x = y]$,

(2) $\forall x, y \in M[d(x, y) = d(y, x)]$,

(3) $\forall x, y, z \in M[d(x, y) \leq d(x, z) + d(z, y)]$.

A metric space is called an ultrametric space if we replace (3) by the stronger (3'):

(3') $\forall x, y, z \in M[d(x, y) \leq \max(d(x, z), d(z, y))]$.

Definition 2.2. Let (M, d) be metric space. Let (x_i) be a sequence in M.

(1) We say that $(x_i)_i$ is a Cauchy sequence whenever we have

$$\forall \varepsilon > 0\, \exists N \in \mathbb{N}\, \forall n, m > N \quad [d(x_n, x_m) < \varepsilon].$$

(2) Let $x \in M$. We say that $(x_i)_i$ converges to x and call x the limit of $(x_i)_i$ whenever we have

$$\forall \varepsilon > 0\, \exists N \in \mathbb{N}\, \forall m > N \quad [d(x, x_m) < \varepsilon].$$

Such a sequence we call convergent. Notation: $\lim_{i \to \infty} x_i = x$.

(3) The metric space (M, d) is called complete whenever each Cauchy sequence converges to an element of M.

Definition 2.3. Let (M_1, d_1), (M_2, d_2) be metric spaces. Let $0 \leq c < 1$. A function f from M_1 to M_2 which satisfies

$$\forall x, y \in M \quad [d_2(f(x), f(y)) \leq c.d_1(x, y)]$$

we call contracting.

Theorem 2.4 (Banach's fixed point theorem). *Let (M, d) be a complete metric space and $f: M \to M$ a contracting function. Then there exists an $x \in M$ such that the following hold:*

(1) $f(x) = x$ (*x is a fixed point of f*),

(2) $\forall y \in M\, [f(y) = y \Rightarrow y = x]$ (*x is unique*).

Definition 2.5 (*Closed subsets*). A subset X of a metric space (M, d) is called closed whenever each Cauchy sequence of elements in X converges to an element of X.

Definition 2.6. The closure $\mathrm{Cl}(X)$ of a subset X of a metric space is the set $\{\lim_{i \to \infty} y_i : \forall i\, [y_i \in X] \wedge (y_i)_i$ is a Cauchy sequence$\}$.

Definition 2.7 (*Compact subsets*). A subset X of a metric space (M, d) is called compact whenever each sequence of elements in X has a convergent subsequence.

Definition 2.8. Let (M, d), (M_1, d_1), (M_2, d_2) be metric spaces.

(1) We define a metric d on the functions in $M_1 \to M_2$ as follows. For $f_1, f_2 \in M_1 \to M_2$

$$d(f_1, f_2) = \sup\{d_2(f_1(x), f_2(x)) : x \in M_1\}.$$

(2) Let

$$\mathcal{P}_{\mathrm{co}}(M) = \{X \subset M : X \text{ is compact and non-empty}\}$$

and

$$\mathcal{P}_{\mathrm{cl}}(M) = \{X \subset M : X \text{ is closed and non-empty}\}.$$

We define a metric d_H on both $\mathcal{P}_{co}(M)$ and $\mathcal{P}_{cl}(M)$, called the Hausdorff distance, as follows. For every $X, Y \in \mathcal{P}_{co}(M)(\in \mathcal{P}_{cl}(M))$

$$d_H(X, Y) = \max\{\sup\{d(x, Y): x \in X\}, \sup\{d(y, X): y \in Y\}\}$$

where $d(x, Z) = \inf\{d(x, z): z \in Z\}$ for every $Z \subset M$, $x \in M$.

(3) We define a metric on the cartesian product $M_1 \times M_2$ as follows:

$$d((x_1, y_1), (x_2, y_2)) = \begin{cases} d_1(x_1, x_2) & x_1 \neq x_2, \\ \frac{1}{2} \cdot d_2(y_1, y_2) & x_1 = x_2. \end{cases}$$

Theorem 2.9. *Let (M, d), (M_1, d_1), (M_2, d_2) be complete (ultra)metric spaces. We have that $M_1 \to M_2$, $\mathcal{P}_{co}(M)$, $\mathcal{P}_{cl}(M)$ and $M_1 \times M_2$ (with the metrics defined above) are complete (ultra)metric spaces.*

In the sequel we sometimes suppress definitions of metrics. We then assume that they are constructed in the standard way outlined above.

3. Syntax and operational semantics

Assume given a (possibly infinite) set of atomic actions B, with typical element b. Let *Proc*, with typical element P, be a set of procedure variables. These two basic sets are used in the following.

Definition 3.1. We define the set of statements $\mathcal{L}$, with typical element s, by the following grammar:

$$s ::= b \mid P \mid s_1; s_2 \mid s_1 + s_2 \mid s_1 \| s_2 \mid [s].$$

A statement s is one of the following six forms:

- an elementary action b;
- a procedure variable P;
- the sequential composition $s_1; s_2$ of statements s_1 and s_2;
- the non-deterministic choice $s_1 + s_2$;
- the concurrent execution $s_1 \| s_2$, modeled by arbitrary interleaving;
- the atomic version $[s]$ of s, modeled by interpreting s as an elementary action.

Assume given a set of states Σ, with typical element σ. Let $Int = B \to \Sigma \to_{partial} \Sigma$ be the set of *interpretations* and let f be a typical element of *Int*. Given an elementary action b and an initial state σ, $f(b)(\sigma)$ (it it exists) is the state after the execution of b in state σ. The set of declarations *Decl* (with typical element d) has as elements functions from $Proc \to \mathcal{L}_g$, where $\mathcal{L}_g$ (the set of *guarded statements*) is defined as follows.

Definition 3.2. We define the set of guarded statements $\mathcal{L}_g$, with typical element g, by the following syntax:

$$g ::= b \mid g; s \mid g_1 + g_2 \mid g_1 \| g_2 \mid [g].$$

Note that $\mathscr{L}_g \subset \mathscr{L}$. Intuitively, a statement s is guarded if all procedure variables occurring in it are preceded by some statement. A program is a triple (f, d, s), where s is a statement, $d \in Decl$ is a declaration for the procedure variables in s and f is an interpretation of the atomic actions. Let *Prog* be the set of programs. In the sequel we sometimes suppress the declaration and interpretation parts of a program; instead of writing (f, d, s) we write just s. The operational semantics for $\mathscr{L}$ is based on a transition relation in the style of [18, 30]. A transition relation describes the steps we can take during a computation. We use a special symbol E, which stands for termination. A step can change the state and the (rest of the) program we have to execute.

Definition 3.3. Let

$$\rightarrow \subseteq (Prog \times \Sigma \times (Prog \cup \{E\}) \times \Sigma)$$

be the smallest relation satisfying (writing $(s, \sigma) \rightarrow (s', \sigma')$ for $(s, \sigma, s', \sigma') \in \rightarrow$ and $(s, \sigma) \rightarrow (E, \sigma')$ for $(s, \sigma, E, \sigma') \in \rightarrow$)

- $(b, \sigma) \rightarrow (E, f(b)(\sigma))$ if $f(b)(\sigma)$ exists,
- $(d(P), \sigma) \rightarrow (s, \sigma') \Rightarrow (P, \sigma) \rightarrow (s, \sigma')$,
- $(d(P), \sigma) \rightarrow (E, \sigma') \Rightarrow (P, \sigma) \rightarrow (E, \sigma')$,
- $(s_1, \sigma_1) \rightarrow (s_2, \sigma_2) \Rightarrow (s_1; s, \sigma_1) \rightarrow (s_2; s, \sigma_2)$
  $(s_1 \| s, \sigma_1) \rightarrow (s_2 \| s, \sigma_2)$
  $(s \| s_1, \sigma_1) \rightarrow (s \| s_2, \sigma_2)$
  $(s + s_1, \sigma_1) \rightarrow (s_2, \sigma_2)$
  $(s_1 + s, \sigma_1) \rightarrow (s_2, \sigma_2)$,
- $(s_1, \sigma_1) \rightarrow (E, \sigma_2) \Rightarrow (s_1; s, \sigma_1) \rightarrow (s, \sigma_2)$
  $(s_1 \| s, \sigma_1) \rightarrow (s, \sigma_2)$
  $(s \| s_1, \sigma_1) \rightarrow (s, \sigma_2)$
  $(s + s_1, \sigma_1) \rightarrow (E, \sigma_2)$
  $(s_1 + s, \sigma_1) \rightarrow (E, \sigma_2)$,
- $(s, \sigma) \rightarrow^* (E, \sigma') \Rightarrow ([s], \sigma) \rightarrow (E, \sigma')$
  (writing $\rightarrow^*$ for the transitive closure of $\rightarrow$).

The last rule takes several transitions together; in order to get a step from $([s], \sigma)$ we analyse sequences of steps from (s, σ). We have the following lemma.

Lemma 3.4. *For all $s \in \mathscr{L}$ and $\sigma \in \Sigma$, the set*

$$\{s' \in \mathscr{L}: \exists \sigma' \in \Sigma\ [(s, \sigma) \rightarrow (s', \sigma')]\}$$

is a finite set.

Please note that the set (for any $s \in \mathscr{L}$ and $\sigma \in \Sigma$) $\{(s', \sigma') \in \mathscr{L} \times \Sigma : (s, \sigma) \rightarrow (s', \sigma')\}$ is in general an infinite set.

We use the transition relation to give an operational semantics; we collect the sequence of states during a computation. Such a sequence can be finite or infinite. We also signal deadlock by a special symbol δ. Deadlock means that from a configuration (s, σ) no transition is possible. This can happen because *Int* contains partial functions. Let Σ^* (Σ^ω) denote the collection of all finite (infinite) words over Σ. Let x be a typical element of Σ^* and let y be a typical element of $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. Let Σ^+ be Σ^* without the empty word and let $\Sigma_\delta^{st} = \Sigma^* \cdot \{\delta\} \cup \Sigma^+ \cup \Sigma^\omega$. Let z be a typical element of Σ_δ^{st}. Put $\mathbf{S} = \Sigma \to \mathcal{P}(\Sigma_\delta^{st})$; the set of functions from Σ to subsets of Σ_δ^{st}. The operational semantics is given in the following.

Definition 3.5. Let $\mathcal{O}: Prog \to \mathbf{S}$ be given by

$$\mathcal{O}(s) = \lambda\sigma.\{\sigma_1 \ldots \sigma_n : (s, \sigma) \to (s_1, \sigma_1) \to \cdots \to (E, \sigma_n)\} \cup$$
$$\{\sigma_1 \ldots \sigma_n \ldots : (s, \sigma) \to (s_1, \sigma_1) \to \cdots \to (s_n, \sigma_n) \to \cdots\} \cup$$
$$\{\sigma_1 \ldots \sigma_n \delta : (s, \sigma) \to (s_1, \sigma_1) \to \cdots \to (s_n, \sigma_n) \nrightarrow\}$$

(writing $(s_n, \sigma_n) \nrightarrow$ for $\forall s, \sigma[(s_n, \sigma_n, s, \sigma) \notin \to \wedge (s_n, \sigma_n, E, \sigma) \notin \to]$).

Note that the operational semantics is not compositional: take for example: $B = \{v := 1, v := 2, v := v + 1\}$, $\Sigma = \mathbb{N}$ (the set of integers). The state records the value of v. Let $f \in Int$ be such that $f(v := 1)(\sigma) = 1, f(v := 2)(\sigma) = 2, f(v := v+1)(\sigma) = \sigma + 1$. Take $s_1 = v := 1;\ v := v + 1$ and $s_2 = v := 1;\ v := 2$. We have $\mathcal{O}(s_1) = \mathcal{O}(s_2) = \lambda\sigma.\{12\}$, but $\mathcal{O}(s_1 \| s_1) = \lambda\sigma.\{1212, 1123\}$ and $\mathcal{O}(s_2 \| s_2) = \lambda\sigma.\{1212, 1122\}$.

The transition system and its derived operational semantics do not take local deadlock (inside atomic brackets) into account. A typical example is the statement $[a; \mathit{fail} + a; b]$. Definition 3.5 gives only the successful computations (i.e. the state changes of $a; b$) and does not consider the possibility $a; \mathit{fail}$. It is possible to extend the transition system with the following special deadlock rules:

- $(b, \sigma) \to (E, \delta)$ if $f(b)(\sigma)$ is undefined;
- $(d(P), \sigma) \to (E, \delta) \Rightarrow (P, \sigma) \to (E, \delta)$;
- $(s_1, \sigma_1) \to (E, \delta) \Rightarrow (s_1; s, \sigma_1) \to (E, \delta)$;
- $(s_1, \sigma) \to (E, \delta) \wedge (s_2, \sigma) \to (E, \delta) \Rightarrow (s_1 \| s_2, \sigma) \to (E, \delta)$
  $(s_1 + s_2, \sigma) \to (E, \delta)$;
- $(s, \sigma) \to^* (E, \delta) \Rightarrow ([s], \delta) \to (E, \delta)$.

Another extension is needed to cope with infinite computations inside guards. We do not consider these two extensions here. For an account of these features we refer to [20].

We turn Σ_δ^{st} into a complete metric space with the help of a prefix operator: Definie for each $z \in \Sigma_\delta^{st}$, $z[n]$ as the prefix of z of length n, if this exists, and $z[n] = z$, otherwise.

Definition 3.6. We define a metric d_{st} on Σ_δ^{st} by putting $d_{st}(z_1, z_2) = 2^{-N}$ if $z_1 \neq z_2$ where $N = \sup\{n : z_1[n] = z_2[n]\}$ and $d_{st}(z_1, z_2) = 0$ if $z_1 = z_2$.

We have the following lemma.

Lemma 3.7. *For any $s \in \mathcal{L}$ and $\sigma \in \Sigma$, $\mathcal{O}(s)(\sigma)$ is a closed set.*

Proof. Take any $s \in \mathcal{L}$ and $\sigma \in \Sigma$. Suppose $(y_i)_i$ is a Cauchy sequence in $\mathcal{O}(s)(\sigma)$. We show that $\lim_{i\to\infty} y_i \in \mathcal{O}(s)(\sigma)$. We only consider the case that $\lim_{i\to\infty} y_i \in \Sigma^\omega$ (otherwise $\lim_{i\to\infty} y_i$ is constant from some moment on). Suppose that $\lim_{i\to\infty} y_i = \sigma_1\sigma_2\ldots$. For all i we have that $y_i \in \mathcal{O}(s)(\sigma)$ and this implies that for any i we can pick sequences $(s_{ij}, \sigma_{ij})_j$ such that

$$(s, \sigma) \to (s_{i1}, \sigma_{i1}) \to (s_{i2}, \sigma_{i2}) \to \cdots$$

and $y_i = \sigma_{i1}\sigma_{i2}\ldots$.

Because $(y_i)_i$ is a Cauchy sequence, an infinite number of σ_{i1} equals σ_1. By Lemma 3.4, there is only a finite number of possibilities for s_{i1}. Hence there exists a $s_1 \in \mathcal{L}$ such that an infinite number of tuples (s_{i1}, σ_{i1}) equals (s_1, σ_1). We can continue this construction and find a sequence of statements (s_i) such that

$$(s, \sigma) \to (s_1, \sigma_1) \to (s_2, \sigma_2) \to \cdots.$$

Hence $y = \sigma_1\sigma_2\ldots \in \mathcal{O}(s)(\sigma)$. □

Next we give an alternative definition for the operational semantics. Lemma 3.4 is used to show that this definition is well-defined. In the proof that the two definitions of $\mathcal{O}$ coincide, we use Lemma 3.7.

From this moment on, we restrict **S** to the set of functions from Σ to the *closed* non-empty subsets of Σ_δ^{st}: $\mathbf{S} = \Sigma \to \mathcal{P}_{cl}(\Sigma_\delta^{st})$. This enables us to assign a metric to **S** in the standard way described in the previous section.

Definition 3.8 (*Alternative definition for $\mathcal{O}$*). Let $\mathcal{O}$: *Prog* $\to$ **S** be the unique fixed point of the contraction $\Delta : (Prog \to \mathbf{S}) \to (Prog \to \mathbf{S})$ which is defined as follows:

$$\Delta(F)(s) = \lambda\sigma.\{\delta : (s, \sigma) \nrightarrow\} \cup$$
$$\{\sigma_1 : (s, \sigma) \to (E, \sigma_1)\} \cup$$
$$\bigcup\{\sigma_1 \cdot F(s_1)(\sigma_1) : (s, \sigma) \to (s_1, \sigma_1)\}.$$

First we show that the definition above is well-defined, i.e that for any F, s, σ we have that $\Delta(F)(s)(\sigma)$ is a *closed* set. Take any F, s, σ and a Cauchy sequence

$(z_i)_i$ in $\Delta(F)(s)(\sigma)$. There exists an infinite subsequence of $(z_{f(i)})_i$ ($f:\mathbb{N}\to\mathbb{N}$ monotonic) in one of the three following sets:

(1) $\{\delta:(s,\sigma)\not\to\}$;

(2) $\{\sigma_1:(s,\sigma)\to(E,\sigma_1)\}$;

(3) $\bigcup\{\sigma_1\cdot F(s_1)(\sigma_1):(s,\sigma)\to(s_1,\sigma_1)\}$.

We consider only the third case. By Lemma 3.4 we know that there is only a finite number of possibilities for s_1. Hence we can pick a σ_1 and a monotonic function $g:\mathbb{N}\to\mathbb{N}$ such that $(z_{g(f(i))})_i$ is an infinite subsequence of $(z_{f(i)})_i$ in $\{\sigma_1\cdot F(s_1)(\sigma_1):(s,\sigma)\to(s_1,\sigma_1)\}$. (Only σ_1 is free.) Because $(z_{g(f(i))})_i$ is an infinite subsequence of a Cauchy sequence, we can find a σ_1 and a monotonic function $h:\mathbb{N}\to\mathbb{N}$ such that $(z_{h(g(f(i)))})_i$ is an infinite sequence in $\{\sigma_1\cdot F(s_1)(\sigma_1):(s,\sigma)\to(s_1,\sigma_1)\}$. Because (by definition of F) $F(s_1)(\sigma_1)$ is a closed set we have that $\{\sigma_1\cdot F(s_1)(\sigma_1):(s,\sigma)\to(s_1,\sigma_1)\}$ is a closed set. Hence the infinite subsequence $(z_{h(g(f(i)))})_i$ of $(z_i)_i$ converges to an element in $\Delta(F)(s)(\sigma)$. So also the whole Cauchy sequence converges to the same element in $\Delta(F)(s)(\sigma)$.

Next we show that the two definitions for $\mathcal{O}$ coincide. By Lemma 3.7 we know that $\mathcal{O}:\mathit{Prog}\to S$ ($\mathcal{O}$ of Definition 3.5). It is not difficult to see that $\Delta(\mathcal{O})=\mathcal{O}$. By Banach's fixed point theorem we have that the two fixed points are the same.

4. Denotational semantics

In this section we define a denotational semantics for $\mathcal{L}$. We call a semantics denotational if it is compositionally defined and treats recursion with the help of fixed points. With each operator in $\mathcal{L}$ we associate a semantic operator. The denotational semantics will be the fixed point of a higher-order operator. The denotational semantics will be based on domains which are metric spaces. These domains are defined as solutions of domain equations.

In the construction of the domains for the denotational semantics we need an operator $\Sigma\,\square$ defined as follows.

Definition 4.1. Let (M,d) be a metric space. We define a metric $\tilde{d}$ on $\Sigma\,\square\,M =^{\text{def}} \Sigma^{\text{st}}_{\delta}\cup\Sigma^{+}\times M$ by putting

- $\tilde{d}(z_1,z_2)=d_{\text{st}}(z_1,z_2)$ if $z_1,z_2\in\Sigma^{\text{st}}_{\delta}$,
- $\tilde{d}((z_1,m_1),(z_2,m_2))$
$$=\begin{cases} d_{\text{st}}(z_1,z_2) & \text{if } z_1,z_2\in\Sigma^{+},\ m_1,m_2\in M,\ z_1\neq z_2\\ 2^{-\mathit{length}(z_1)}\cdot d(m_1,m_2) & \text{if } z_1,z_2\in\Sigma^{+},\ m_1,m_2\in M,\ z_1=z_2,\end{cases}$$
- $\tilde{d}(z_1,(z_2,m))$
$$=\begin{cases} d_{\text{st}}(z_1,z_2) & \text{if } z_1\in\Sigma^{\text{st}}_{\delta},\ z_2\in\Sigma^{+},\ m\in M,\ z_1\neq z_2\\ 2^{-\mathit{length}(z_1)} & \text{if } z_1\in\Sigma^{\text{st}}_{\delta},\ z_2\in\Sigma^{+},\ m\in M,\ z_1=z_2,\end{cases}$$
- $\tilde{d}((z_1,m),z_2)=\begin{cases} d_{\text{st}}(z_1,z_2) & \text{if } z_2\in\Sigma^{\text{st}}_{\delta},\ z_1\in\Sigma^{+},\ m\in M,\ z_1\neq z_2\\ 2^{-\mathit{length}(z_1)} & \text{if } z_2\in\Sigma^{\text{st}}_{\delta},\ z_1\in\Sigma^{+},\ m\in M,\ z_1=z_2.\end{cases}$

We have that $(\Sigma \square M, \tilde{d})$ is a complete ultrametric space if (M, d) is a complete ultrametric space. We briefly recall the notion of a (metric) domain equation. The general form of such an equation is $\mathbf{P} = F(\mathbf{P})$ or, more precisely, $(\mathbf{P}, s) \cong F((\mathbf{P}, d))$, where the mapping F maps metric spaces to metric spaces. Under certain conditions, we can find unique solutions (up to isometry) in the category of complete metric spaces. We have no room to discuss details (see references after Definition 4.2). For our purposes, it is sufficient to know that $\mathbf{P} = \Sigma \to \mathcal{P}_{co}(\Sigma \square \mathbf{P})$ has a complete ultrametric space as solution. Formally, the metric on $\Sigma \to \mathcal{P}_{co}(\Sigma \square \mathbf{P})$ is derived from the metric on $M_1 = \mathbf{P}$, $M_2 = \Sigma \square M_1$, $M_3 = \mathcal{P}_{co}(M_2)$ and $M_4 = \Sigma \to M_3$.

Definition 4.2. Let $\mathbf{P}$ be the unique complete ultrametric space that satisfies

$$\mathbf{P} = \Sigma \to \mathcal{P}_{co}(\Sigma \square \mathbf{P}).$$

Elements of $\mathbf{P}$ are called processes. Let p be a typical element of $\mathbf{P}$. Given an initial state σ, $p(\sigma)$ is a (compact) set. Elements of this set are either in Σ^{st}_{δ} or in $\Sigma^+ \times \mathbf{P}$. An element in Σ^+ (Σ^ω) can be seen as a (non) terminating computation, an element in $\Sigma^* \cdot \{\delta\}$ as a computation ending in deadlock. An element in $\Sigma^+ \times \mathbf{P}$ can be viewed as a terminating computation which has a resumption; after the computation (which is finite), it turns itself into another process.

The way in which we solved the domain equation does not completely follow the usual pattern of solving domain equations as described in [10] or [1]. However, when we consider the two equations

$$\mathbf{P} = \Sigma \to \mathcal{P}_{co}(\mathbf{Q}), \qquad \mathbf{Q} = \Sigma \cup \{\delta\} \cup \Sigma \times \mathbf{Q} \cup \Sigma \times \mathbf{P},$$

we can apply the usual pattern and obtain an isometric domain.

For each syntactic operator in $\mathcal{L}$ we define a semantic operator. The semantic operators corresponding with ;, + and $\|$ will be of type $\mathbf{P} \times \mathbf{P} \to \mathbf{P}$ and the semantic operator corresponding to $[\cdot]$ will be of type $\mathbf{P} \to \mathbf{P}$.

Definition 4.3. The operators $\tilde{;}$, $\tilde{+}$, $\tilde{\|}$, $\mathbf{P} \times \mathbf{P} \to \mathbf{P}$ and the operator stream, $\mathbf{P} \to \mathbf{P}$ are defined as follows. Let

$$p_1 \tilde{+} p_2 = \lambda\sigma.(p_1(\sigma) \cup p_2(\sigma))$$

and let $\tilde{;}$, $\tilde{\|}$ be the unique fixed points of the contractions $\Phi_{;}, \Phi_{\|} : (\mathbf{P} \times \mathbf{P} \to \mathbf{P}) \to (\mathbf{P} \times \mathbf{P} \to \mathbf{P})$ that are defined as follows (F ranges over $\mathbf{P} \times \mathbf{P} \to \mathbf{P}$)

$$\Phi_{;}(F)(p_1, p_2) = \lambda\sigma.\{z : z \in p_1(\sigma) \wedge z \in \Sigma^\omega \cup \Sigma^* \cdot \{\delta\}\} \cup$$
$$\{(z, p_2) : z \in p_1(\sigma) \wedge z \in \Sigma^+\} \cup$$
$$\{(z, F(p, p_2)) : (z, p) \in p_1(\sigma)\},$$

$$\Phi_{\|}(F)(p_1, p_2) = \Phi_{;}(F)(p_1, p_2) \tilde{+} \Phi_{;}(F)(p_2, p_1).$$

and let stream $\mathbf{P} \to \mathbf{P}$ be the unique fixed point of the contraction $\Phi_{stream} : (\mathbf{P} \to \mathbf{P}) \to (\mathbf{P} \to \mathbf{P})$ that is defined by

$$\Phi_{stream}(F)(p) = \lambda\sigma.\{z \in \Sigma_\delta^{st} : z \in p(\sigma)\} \cup \{x\sigma' z : (x\sigma', p') \in p(\sigma) \wedge z \in F(p')(\sigma')\}.$$

We give some explanation for the *stream* operation. First note that for all processes p and states σ we have that $stream(p)(\sigma) \subseteq \Sigma_\delta^{st}$. Hence we can say that the operator *stream* removes the tree-like structure of the process. It also removes the interleaving points of a process. Processes allow for interleaving; an element of $p(\sigma)$ can be of the form (x, p'). After computation x it turns itself into p'. Before starting the computations in p', other processes can do some computations. Accordingly we say that between x and p' we have an interleaving point. The operator *stream* removes these points by passing the final state of the computation x as argument to the process p'.

In the sequel we use a left-merge operator.

Definition 4.4. Define $\tilde{\lfloor\!\lfloor} : \mathbf{P} \times \mathbf{P} \to \mathbf{P}$ by $\tilde{\lfloor\!\lfloor} = \Phi_{\tilde{;}}(\tilde{\|})$.

We often write ;, +, $\|$, $\lfloor\!\lfloor$ rather than $\tilde{;}$, $\tilde{+}$, $\tilde{\|}$, $\tilde{\lfloor\!\lfloor}$ if no confusion is possible. We have the following lemma.

Lemma 4.5. *For all* $p_1, p_2, p_1', p_2' \in \mathbf{P}$ *and for all* $\mathrm{op} \in \{;, \|, \lfloor\!\lfloor, +\}$ *we have*

$$d(p_1 \,\mathrm{op}\, p_2, p_1' \,\mathrm{op}\, p_2') \leq \max\{d(p_1, p_1'), d(p_2, p_2')\}$$

and

$$d(stream(p_1), stream(p_2)) \leq d(p_1, p_2).$$

A proof of a very similar lemma is given in [10]. Now we can define a denotational semantics for $\mathscr{L}$ in the following.

Definition 4.6. Let $\mathscr{D} : Prog \to \mathbf{P}$ be the unique fixed point of the contraction $\Phi : (Prog \to \mathbf{P}) \to (Prog \to \mathbf{P})$ which is defined inductively as follows.

- $\Phi(F)(b) = \lambda\sigma.\begin{cases}\{f(b)(\sigma)\} & \text{if } f(b)(\sigma) \text{ exists} \\ \{\delta\} & \text{otherwise,}\end{cases}$
- $\Phi(F)(P) = \Phi(F)(d(P))$,
- $\Phi(F)(s_1; s_2) = \Phi(F)(s_1); F(s_2)$,
- $\Phi(F)(s_1 + s_2) = \Phi(F)(s_1) + \Phi(F)(s_2)$,
- $\Phi(F)(s_1 \| s_2) = \Phi(F)(s_1) \| \Phi(F)(s_2)$,
- $\Phi(F)([s]) = stream(\Phi(F)(s))$.

5. Relation between the operational and denotational semantics

The operational semantics $\mathcal{O}$ delivers linear time objects (for a given state σ, $\mathcal{O}(s)(\sigma) \subseteq \Sigma_\delta^{st}$) whereas the denotational semantics $\mathcal{D}$ delivers branching time objects in **P**. We define a restriction operator *restr* which will link $\mathcal{O}$ and $\mathcal{D}$: given a process p and an initial state σ, it delivers certain "paths" in the process p. A path will be an element of Σ_δ^{st}. In the next definition we use the operator *last* which takes the last element of a word in Σ^+.

Definition 5.1. Let $restr: \mathbf{P} \to \mathbf{S}$ be the unique fixed point of the contraction

$$\Gamma: (\mathbf{P} \to \mathbf{S}) \to (\mathbf{P} \to \mathbf{S})$$

which is given by $\Gamma(F)(p)(\sigma) = \{\delta\}$ if $p(\sigma) \subset \Sigma^\omega \cup \Sigma^* \cdot \{\delta\}$ and

$$\Gamma(F)(p)(\sigma) = \mathrm{Cl}(\{last(x): x \in \Sigma^+ \wedge x \in p(\sigma)\} \cup \bigcup \{last(x) \cdot F(p')(last(x)): (x, p') \in \Sigma^+ \times \mathbf{P} \wedge (x, p') \in p(\sigma)\})$$

otherwise.

We have the following theorem.

Theorem 5.2. $\mathcal{O} = restr \circ \mathcal{P}$.

In order to prove this theorem, we will define an intermediate semantics $\mathcal{I}$. It is called intermediate because it serves as an intermediate semantics between $\mathcal{O}$ and $\mathcal{D}$: it is defined with the help of a transition system (like the operational semantics) and it delivers tree-like objects (like the denotational semantics). An essential further property of the intermediate semantics (compared to the operational semantics) is that it keeps the intermediate states in the computation. This facilitates the proof of the theorem which we prove in two steps. First we show that $\mathcal{I} = \mathcal{D}$ and secondly we show that $\mathcal{O} = restr \circ \mathcal{I}$.

We prove $\mathcal{I} = \mathcal{D}$ by showing that $\mathcal{D}$ is a fixed point of the defining contraction of $\mathcal{I}$, and hence, by Banach's theorem, we have that $\mathcal{I} = \mathcal{D}$. We give the transition system for the intermediate semantics $\mathcal{I}$ in the following.

Definition 5.3. Let

$$\to \subseteq Prog \times \Sigma \times \Sigma_\delta^{st} \times (Prog \cup \{E\}) \times (\Sigma \cup \{\delta\})$$

be the smallest relation satisfying (writing $(s, \sigma) \xrightarrow{z} (s', \sigma')$ for $(s, \sigma, z, s', \sigma') \in \to$, $(s, \sigma) \xrightarrow{z} (E, \sigma')$ for $(s, \sigma, z, E, \sigma') \in \to$, and $(s, \sigma) \xrightarrow{z} (E, \delta)$ for $(s, \sigma, z, E, \delta) \in \to$)

- $(b, \sigma) \xrightarrow{f(b)(\sigma)} (E, f(b)(\sigma))$ if $f(b)(\sigma) \in \Sigma$,
- $(b, \sigma) \xrightarrow{\delta} (E, \delta)$ if $f(b)(\sigma)$ is undefined,
- $(d(P), \sigma) \xrightarrow{z} (s, \sigma') \Rightarrow (P, \sigma) \xrightarrow{z} (s, \sigma')$,

- $(d(P), \sigma) \xrightarrow{z} (E, \sigma') \Rightarrow (P, \sigma) \xrightarrow{z} (E, \sigma')$,
- $(d(P), \sigma) \xrightarrow{z} (E, \delta) \Rightarrow (P, \sigma) \xrightarrow{z} (E, \delta)$,
- $(s_1, \sigma_1) \xrightarrow{z} (s_2, \sigma_2) \Rightarrow (s_1; s, \sigma_1) \xrightarrow{z} (s_2; s, \sigma_2)$
  $(s_1 \| s, \sigma_1) \xrightarrow{z} (s_2 \| s, \sigma_2)$
  $(s \| s_1, \sigma_1) \xrightarrow{z} (s \| s_2, \sigma_2))$
  $(s + s_1, \sigma_1) \xrightarrow{z} (s_2, \sigma_2)$
  $(s_1 + s, \sigma_1) \xrightarrow{z} (s_2, \sigma_2)$,
- $(s_1, \sigma_1) \xrightarrow{z} (E, \sigma_2) \Rightarrow (s_1; s, \sigma_1) \xrightarrow{z} (s, \sigma_2)$
  $(s_1 \| s, \sigma_1) \xrightarrow{z} (s, \sigma_2)$
  $(s \| s_1, \sigma_1) \xrightarrow{z} (s, \sigma_2)$
  $(s + s_1, \sigma_1) \xrightarrow{z} (E, \sigma_2)$
  $(s_1 + s, \sigma_1) \xrightarrow{z} (E, \sigma_2)$,
- $(s_1, \sigma_1) \xrightarrow{z} (E, \delta) \Rightarrow (s_1; s, \sigma_1) \xrightarrow{z} (E, \delta)$
  $(s_1 \| s, \sigma_1) \xrightarrow{z} (E, \delta)$
  $(s \| s_1, \sigma_1) \xrightarrow{z} (E, \delta)$
  $(s + s_1, \sigma_1) \xrightarrow{z} (E, \delta)$
  $(s_1 + s, \sigma_1) \xrightarrow{z} (E, \delta)$,
- $(s_1, \sigma_1) \xrightarrow{z_1} (s_2, \sigma_2) \xrightarrow{z_2} \cdots (s_n, \sigma_n)$
  $\xrightarrow{z_n} (E, \sigma) \Rightarrow ([s_1], \sigma_1) \xrightarrow{z_1 \cdots z_n} (E, \sigma)$,
- $(s_1, \sigma_1) \xrightarrow{z_1} (s_2, \sigma_2) \xrightarrow{z_2} \cdots (s_n, \sigma_n)$
  $\xrightarrow{z_n} (E, \delta) \Rightarrow ([s_1], \sigma_1) \xrightarrow{z_1 \cdots z_n} (E, \delta)$,
- $(s_1, \sigma_1) \xrightarrow{z_1} (s_2, \sigma_2) \xrightarrow{z_2} \cdots (s_n, \sigma_n) \xrightarrow{z_n} \cdots \Rightarrow ([s_1], \sigma_1) \xrightarrow{z_1 \cdots z_n \cdots} (E, \delta)$.

Note that we have defined two transition relations; one in Definition 3.3 and the other in Definition 5.3. The second relation is always written with a superscript. The following lemma holds.

Lemma 5.4

$$\exists z\,[(s, \sigma) \xrightarrow{z} (s', \sigma')] \Leftrightarrow (s, \sigma) \to (s', \sigma'),$$

$$\exists z\,[(s, \sigma) \xrightarrow{z} (E, \sigma')] \Leftrightarrow (s, \sigma) \to (E, \sigma').$$

It follows that

$$\neg\exists z, s', \sigma' \quad [(s,\sigma) \xrightarrow{z} (s',\sigma') \vee (s,\sigma) \xrightarrow{z} (E,\sigma')] \Leftrightarrow (s,\sigma) \nrightarrow.$$

Next we give the intermediate semantics.

Definition 5.5. Let $\mathscr{I} : \mathit{Prog} \to \mathbf{P}$ be the unique fixed point of the contraction $\Psi : (\mathit{Prog} \to \mathbf{P}) \to (\mathit{Prog} \to \mathbf{P})$ which is defined as follows:

$$\begin{aligned}\Psi(F)(s) = \lambda\sigma.&\{z : (s,\sigma) \xrightarrow{z} (E,\sigma')\} \cup \\ &\{z : (s,\sigma) \xrightarrow{z} (E,\delta)\} \cup \\ &\{(z, F(s')) : (s,\sigma) \xrightarrow{z} (s',\sigma')\}.\end{aligned}$$

In Appendix A we show that Ψ is well-defined (i.e. for any F, s and σ we have that $\Psi(F)(s)(\sigma)$ is a compact set). We provde a lemma with properties of the defining contraction Ψ of $\mathscr{I}$.

Lemma 5.6

(1) $\Psi(\mathscr{D})(b) = \mathscr{D}(b)$,

(2) $\Psi(\mathscr{D})(P) = \Psi(\mathscr{D})(d(P))$,

(3) $\Psi(\mathscr{D})(s_1; s_2) = \Psi(\mathscr{D})(s_1); \mathscr{D}(s_2)$,

(4) $\Psi(\mathscr{D})(s_1 + s_2) = \Psi(\mathscr{D})(s_1) + \Psi(\mathscr{D})s_2)$,

(5) $\Psi(\mathscr{D})(s_1 \| s_2) = \Psi(\mathscr{D})(s_1) \mathbin{\mathbb{L}} \mathscr{D}(s_2) + \Psi(\mathscr{D})(s_2) \mathbin{\mathbb{L}} \mathscr{D}(s_1)$,

(6) $\Psi(\mathscr{D})([s]) \in \mathbf{P} \cap \mathbf{S}$,

(7) $$\begin{aligned}\Psi(\mathscr{D})([s]) = \lambda\sigma.&\{z : (s,\sigma) \xrightarrow{z} (E,\sigma')\} \cup \\ &\{z : (s,\sigma) \xrightarrow{z} (E,\delta)\} \cup \\ &\bigcup \{z \cdot \Psi(\mathscr{D})([s'])(\sigma') : (s,\sigma) \xrightarrow{z} (s',\sigma')\}.\end{aligned}$$

We give some details of the proof of case (5).

$$\begin{aligned}\Psi(\mathscr{D})(s_1 \| s_2) = \lambda\sigma.&\{z : (s_1 \| s_2, \sigma) \xrightarrow{z} (E,\sigma')\} \cup \\ &\{z : (s_1 \| s_2, \sigma) \xrightarrow{z} (E,\delta)\} \cup \\ &\{(z, \mathscr{D}(s')) : (s_1 \| s_2, \sigma) \xrightarrow{z} (s',\sigma')\}.\end{aligned}$$

Because a transition from $(s_1 \| s_2, \sigma)$ never yields a configuration of the form (E, σ') we have that

$$\Psi(\mathscr{D})(s_1 \| s_2) = \lambda\sigma.\{z : (s_1 \| s_2, \sigma) \xrightarrow{z} (E, \delta)\} \cup \{(z, \mathscr{D}(s')) : (s_1 \| s_2, \sigma) \xrightarrow{z} (s', \sigma')\}.$$

By properties of the transition system we have $(s_1 \| s_2, \sigma) \xrightarrow{z} (s', \sigma')$ if and only if

$$\exists s_1' \quad [(s_1, \sigma \xrightarrow{z} (s_1', \sigma') \wedge s' = s_1' \| s_2]$$

or

$$\exists s_2' \quad [(s_2, \sigma) \xrightarrow{z} (s_2', \sigma') \wedge s' = s_1 \| s_2']$$

or

$$(s_1, \sigma) \xrightarrow{z} (E, \sigma') \wedge s' = s_2$$

or

$$(s_2, \sigma) \xrightarrow{z} (E, \sigma') \wedge s' = s_1.$$

By similar properties we have $(s_1 \| s_2, \sigma) \xrightarrow{z} (E, \delta)$ if and only if $(s_1, \sigma) \xrightarrow{z} (E, \delta)$ or $(s_2, \sigma) \xrightarrow{z} (E, \delta)$. Hence

$$\begin{aligned}\Psi(\mathscr{D})(s_1 \| s_2) = \lambda\sigma.&\{z : (s_1, \sigma) \xrightarrow{z} (E, \delta)\} \cup \\ &\{z : (s_2, \sigma) \xrightarrow{z} (E, \delta)\} \cup \\ &\{z, \mathscr{D}(s_1' \| s_2)) : (s_1, \sigma) \xrightarrow{z} (s_1', \sigma')\} \cup \\ &\{(z, \mathscr{D}(s_1 \| s_2')) : (s_2, \sigma) \xrightarrow{z} (s_2', \sigma')\} \cup \\ &\{(z, \mathscr{D}(s_2)) : (s_1, \sigma) \xrightarrow{z} (E, \sigma')\} \cup \\ &\{(z, \mathscr{D}(s_1)) : (s_2, \sigma) \xrightarrow{z} (E, \sigma')\}.\end{aligned}$$

Rearranging and using the compositionality of $\mathscr{D}$ we obtain

$$\begin{aligned}\Psi(\mathscr{D})(s_1 \| s_2) = \lambda\sigma.&\{z : (s_1, \sigma) \xrightarrow{z} (E, \delta)\} \cup \\ &\{(z, \mathscr{D}(s_1') \| \mathscr{D}(s_2)) : (s_1, \sigma) \xrightarrow{z} (s_1', \sigma')\} \cup \\ &\{(z, \mathscr{D}(s_2)) : (s_1, \sigma) \xrightarrow{z} (E, \sigma')\} \cup \\ &\{z : (s_2, \sigma) \xrightarrow{z} (E, \delta)\} \cup \\ &\{(z, \mathscr{D}(s_1) \| \mathscr{D}(s_2')) : (s_2, \sigma) \xrightarrow{z} (s_2', \sigma')\} \cup \\ &\{(z, \mathscr{D}(s_1)) : (s_2, \sigma) \xrightarrow{z} (E, \sigma')\}\end{aligned}$$

and this equals

$$\Psi(\mathscr{D})(s_1 \| s_2) = \Psi(\mathscr{D})(s_1) \mathbin{\lfloor\!\lfloor} \mathscr{D}(s_2) + \Psi(\mathscr{D})(s_2) \mathbin{\lfloor\!\lfloor} \mathscr{D}(s_1).$$

Lemma 5.7. $\Psi(\mathscr{D}) = \mathscr{D}$.

Proof. We show that for all $s \in \mathscr{L}$

$$d(\Psi(\mathscr{D})(s), \mathscr{D}(s)) \leq \tfrac{1}{2} \cdot d(\Psi(\mathscr{D}), \mathscr{D}).$$

This implies that $d(\Psi(\mathscr{D}), \mathscr{D}) \leq \frac{1}{2} \cdot d(\Psi(\mathscr{D}), \mathscr{D})$, i.e. $d(\Psi(\mathscr{D}), \mathscr{D}) = 0$, i.e. $\Psi(\mathscr{D}) = \mathscr{D}$.

We first prove it for $g \in \mathscr{L}_g$. We use structural induction on the elements of $\mathscr{L}_g$. We give only the cases g; s, $[g]$:

$(g; s)$ $d(\Psi(\mathscr{D})(g; s), \mathscr{D}(g; s))$

$= d(\Psi(\mathscr{D})(g); \mathscr{D}(s), \mathscr{D}(g); \mathscr{D}(s))$ (Lemma 5.6)

$\leq \max\{d(\Psi(\mathscr{D})(g), \mathscr{D}(g)), d(\mathscr{D}(s), \mathscr{D}(s))\}$ (Lemma 4.5),

$d(\Psi(\mathscr{D})(g), \mathscr{D}(g)) \leq \frac{1}{2} \cdot d(\Psi(\mathscr{D}), \mathscr{D})$ (induction),

$([g])$ $d(\Psi(\mathscr{D})([g]), \mathscr{D}([g]))$

$\leq \max\{d(\Psi(\mathscr{D})([g]), \mathit{stream}(\Psi(\mathscr{D})(g))),$

$\quad d(\mathit{stream}(\Psi(\mathscr{D})(g)), \mathscr{D}([g]))\}$ (d is an ultrametric).

We show that

(1) $d(\Psi(\mathscr{D})([g]), \mathit{stream}(\Psi(\mathscr{D})(g))) \leq \frac{1}{2} \cdot d(\Psi(\mathscr{D}), \mathscr{D})$,

(2) $d(\mathit{stream}(\Psi(\mathscr{D})(g)), \mathscr{D}([g])) \leq \frac{1}{2} \cdot d(\Psi(\mathscr{D}), \mathscr{D})$.

(1) $\Psi(\mathscr{D})([g]) = \lambda\sigma.\{z : (g, \sigma) \xrightarrow{z} (E, \sigma')\} \cup$

$\{z : (g, \sigma) \xrightarrow{z} (E, \delta)\} \cup$

$\bigcup \{z \cdot \Psi(\mathscr{D})([s'])(\sigma') : (g, \sigma) \xrightarrow{z} (s', \sigma')\}$ (Lemma 5.6).

On the other hand, we have

$\mathit{stream}(\Psi(\mathscr{D})(g)) = \mathit{stream}(\lambda\sigma.\{z : (g, \sigma) \xrightarrow{z} (E, \sigma')\} \cup$

$\{z : (g, \delta) \xrightarrow{z} (E, \delta)\} \cup$

$\{(z, \mathscr{D}(s')) : (g, \sigma) \xrightarrow{z} (s', \sigma')\})$

(definition of Ψ)

$= \lambda\sigma.\{z : (g, \sigma) \xrightarrow{z} (E, \sigma')\} \cup$

$\{z : (g, \sigma) \xrightarrow{z} (E, \delta)\} \cup$

$\bigcup \{z \cdot \mathit{stream}(\mathscr{D})(s')(\sigma')) : (g, \sigma) \xrightarrow{z} (s', \sigma')\}$ (definition of *stream*)

$= \lambda\sigma.\{z : (g, \sigma) \xrightarrow{z} (E, \sigma')\} \cup$

$\{z : (g, \sigma) \xrightarrow{z} (E, \delta)\} \cup$

$\bigcup \{z \cdot \mathscr{D}([s'])) : (g, \sigma) \xrightarrow{z} (s', \sigma')\}$

so

$$d(\Psi(\mathscr{D})([g]), stream(\Psi(\mathscr{D})(g))) \leqslant \tfrac{1}{2} \cdot d(\Psi(\mathscr{D}), \mathscr{D})$$

since $z = y\sigma'$ is not equal to the empty word (hence the factor $\frac{1}{2}$). Note that the last step does not use the induction hypothesis.

(2) $d(stream(\Psi(\mathscr{D})(g)), \mathscr{D}([g])) = d(stream(\Psi(\mathscr{D})(g)), stream(\mathscr{D}(g)))$

$\leqslant d(\Psi(\mathscr{D})(g), \mathscr{D}(g))$ (Lemma 4.5)

$\leqslant \frac{1}{2} \cdot d(\Psi(\mathscr{D}), \mathscr{D})$ (induction hypothesis).

Secondly, we extend $\mathscr{L}_g$ to $\mathscr{L}$. We use structural induction on the elements of $\mathscr{L}$. All cases are the same as for $\mathscr{L}_g$, except for P (which is not present in the guarded case).

(P) By Lemma 5.6 we have $\Psi(\mathscr{D})(P) = \Psi(\mathscr{D})(d(P))$ and by the definition of $\mathscr{D}$ we have $\mathscr{D}(P) = \mathscr{D}(d(P))$. Hence

$$d(\Psi(\mathscr{D})(P), \mathscr{D}(P)) = d(\Psi(\mathscr{D})(d(P)), \mathscr{D}(d(P)))$$

$$\leqslant \tfrac{1}{2} \cdot d(\Psi(\mathscr{D}), \mathscr{D}) \quad (d(P) \text{ is a guarded statement}).$$

By Bananch's fixed point theorem we have the following.

Corollary 5.8. $\mathscr{I} = \mathscr{D}$.

Lemma 5.9. $\mathscr{O} = restr \circ \mathscr{I}$.

Proof. We show that

$$\Delta(restr \circ \mathscr{I}) = restr \circ \mathscr{I}$$

where Δ is the defining contraction of $\mathscr{O}$. By the definition of Δ, $\Delta(restr \circ \mathscr{I})(s)(\sigma) = \{\delta\}$ if $(s, \sigma) \nrightarrow$. Because $(s, \sigma) \nrightarrow$ implies

$$\neg \exists z, s', \sigma' \quad [(s, \sigma) \xrightarrow{z} (s', \sigma') \vee (s, \sigma) \xrightarrow{z} (E, \sigma')],$$

we have by definition of $\mathscr{I}$ that

$$\mathscr{I}(s)(\sigma) \subset \Sigma^{\omega} \cup \Sigma^* \cdot \{\delta\}$$

i.e. $(restr \circ \mathscr{I})(s)(\sigma) = \{\delta\}$. Now assume that there are transitions possible from (s, σ):

$$\Delta(restr \circ \mathscr{I})(s)(\sigma) = \{\sigma' : (s, \sigma) \rightarrow (E, \sigma')\} \cup$$

$$\bigcup \{\sigma' \cdot (restr \circ \mathscr{I})(s')(\sigma') : (s, \sigma) \rightarrow (s', \sigma')\}$$

(because $restr \circ \mathscr{I} \in Prog \rightarrow \mathbf{S}$ we have that $\Delta(restr \circ \mathscr{I})(s)(\sigma)$ is a closed set by

definition of Δ (cf. the justification for Definition 3.8))

$$= \mathrm{Cl}\Big(\{\sigma \mid : (s, \sigma) \to (E, \sigma')\} \cup \bigcup \{\sigma' \cdot (restr \circ \mathcal{I})(s')(\sigma') : (s, \sigma) \to (s', \sigma')\} \Big)$$

$$= \mathrm{Cl}\Big(\{last(x \cdot \sigma') : (s, \sigma) \xrightarrow{x\sigma'} (E, \sigma') \wedge x \in \Sigma^*\} \cup \bigcup \{last(x \cdot \sigma') \cdot (restr \circ \mathcal{I})(s')(last(x \cdot \sigma') : (s, \sigma) \xrightarrow{x\sigma'} (s', \sigma')\} \Big)$$

$$= (restr \circ \mathcal{I})(s)(\sigma).$$

6. Concurrent Prolog

In this section we apply the framework of the previous sections. We choose a set of elementary actions, a set of procedure variables, a set of states and an interpretation function in such a way that we obtain a denotational and an operational semantics for Concurrent Prolog. A Concurrent Prolog program is "translated" to an element of *Prog*.

We first introduce the language Concurrent Prolog (CP) in an informal way. The reader not familiar with CP should consult [35], the paper which introduces the concepts of CP.

Let a be a typical element of the set of atoms *Atom*. Atoms are built up in the usual way from constants, variables, functors and predicate symbols. In CP there is a special functor ? of arity 1 which is called the read-only functor. The paper [33] signals some problems with the interpretation of the read-only functor in [35]. Saraswat gives in [33] an alternative interpretation for the read-only functor (input-only functor in his terms). This interpretation is an extension of normal unification. We take over his interpretation. In this section we do not give a formal definition of the extended unification. We provide an appendix in which we give the details. The rest of the paper can be read without knowledge of the exact details. We denote the extended unification function by $mgu_?$. It is a partial function on $Atom \times Atom$. If it is defined it delivers a substitution.

A CP program is a finite set of elements (called clauses) of the following form:

$$c \leftarrow a_1 \wedge \cdots \wedge a_n \mid a_{n+1} \wedge \cdots \wedge a_m.$$

Both n, m can be 0. The bar $\mid$ is called the commit operator, a the head, $a_1 \wedge \cdots \wedge a_n$ the guard and $a_{n+1} \wedge \cdots \wedge a_m$ the body of the clause. If $n = 0$ we have an empty guard and if $n = m$ we have an empty body. Let *Clause* be the set of clauses and let c be a typical element of *Clause*. Besides a finite set of clauses, we also have a goal which is of the form $\bar{a}_1 \wedge \cdots \wedge \bar{a}_k$.

If $k = 0$ we say that the goal is empty. The (interleaved) execution of a CP program goes as follows. We execute the goal given the identity substitution. The execution of a goal given a (current) substitution means that we try to resolve all the atoms in the goal until the goal is empty. If the goal is empty we return a substitution. In order to resolve an atom we unify it with the head of a clause (taking the current substitution into account) and we try to execute the guard (given the "new" substitution resulting from the unification). The unification and the execution of the guard do not yet influence the current substitution. Only after the guard becomes empty we commit: we do not consider alternatives for this clause anymore and we replace the atom by the body of the clause and update the current substitution. The execution model described here is an interleaving model. We do not consider here a parallel model where we have truly parallel processes. In such a model we also would have to check if a substitution delivered by the execution of a guard matches with the current substitution.

We introduce disjoint sets of variables. This is done for technical reasons. During the process described above we replace atoms by bodies of clauses. In order to avoid clashes of variables, every time we rewrite an atom we "replace" the variables in the clause by new ones. This is intuitively correct because clauses are assumed to be universally quantified. Therefore we partition the set of variables *Var* into infinite disjunct subsets Var_α, where α ranges over $\mathbb{N}^*$, the set of finite words of integers. Assume injections $\alpha : Var_\varepsilon \to Var_\alpha$ (and their natural extensions to elements of *Atom*). Now we choose our basic sets; take Σ the set of substitutions, $B = Atom \times Atom$ and $Proc = Atom \times \mathbb{N}^*$. A pair (a, α) in *Proc* specifies that we have to rewrite the atom a with a clause of the program in which the variables are taken from Var_α. Take

$$f(a_1, a_2)(\sigma) = mgu_?(a_1, \sigma(a_2)) \circ \sigma$$

if $mgu_?(a_1, \sigma(a_2))$ is defined and is undefined otherwise. The composition $\circ$ is the usual composition of substitutions (see for example [2]).

Fix a CP program and a goal. We assume that all variables in the program and in the goal are taken from Var_ε. We define a function

$$stm : Clause \times Proc \to \mathscr{L}_g$$

by

$$stm(\bar{a} \leftarrow a_1 \wedge \cdots \wedge a_n \,|\, a_{n+1} \wedge \cdots \wedge a_m, (a, \alpha))$$
$$= [(\alpha(\bar{a}), a); (\alpha(a_1), \alpha \cdot 1) \| \cdots \| (\alpha(a_n), \alpha \cdot n)];$$
$$(\alpha(a_{n+1}), \alpha \cdot n + 1) \| \cdots \| (\alpha(a_m), \alpha \cdot m).$$

Suppose the set of clauses is $\{c_1, \ldots, c_k\}$. Define

$$(*) \qquad d(P) = stm(c_1, P) + \cdots + stm(c_k, P).$$

Assume the goal is $\bar{a}_1 \wedge \cdots \wedge \bar{a}_k$. Take

$$s = (\bar{a}_1, 1) \| \cdots \| (\bar{a}_k, k).$$

Some explanation is necessary. Execution of the goal consists of parallel execution of the k procedure variables $(\bar{a}_1, 1), \ldots, (\bar{a}_k, k)$. When we call *stm* on a clause c and a pair (a, α) it considers what has to be done in order to rewrite atom a with clause c in which we have to take the variables from Var_α. Suppose $c = \bar{a} \leftarrow a_1 \wedge \cdots \wedge a_n \mid a_{n+1} \wedge \cdots \wedge a_m$. First we unify a with $\bar{a}$ (the head of clause c). Because we have to take variables from Var_α we rename the variables in $\bar{a}$ with the operator α; this results in the pair $(\alpha(\bar{a}), a)$. After this unification, we have to execute the guard of the clause c, i.e. $a_1 \wedge \cdots \wedge a_n$. We can execute all the atoms (in which the variables are renamed by α) in parallel. In order to avoid clashes of variables, we specify that if $\alpha(a_i)$ is rewritten by a clause, variables in that clause are to be taken from $Var_{\alpha \cdot i}$. The resolving of the guard and the unification is not (yet) allowed to influence other computations. This is modeled by considering them to be an elementary action by placing $[\cdot]$ around the unification and the guard. After the execution of the guard, we continue with the execution of the body; again with the renaming and the specification of sets of variables.

Now we can also have a better understanding of the operator *restr* (see Definition 5.1). Alternative clauses are joined together by the + operator in (*). Semantically this means that they are alternative computations. As indicated above, the grainsize (the computations between two interleaving points) is the computation of guards. If there is a terminating guard computation, it can be chosen; deadlocking and/or non-terminating computations of guards need no longer be considered. If there are only deadlocking and/or non-terminating computations we never reach the commit and we have deadlock.

There is no explicit communication in the language $\mathscr{L}$. The reader might wonder how the communication of Concurrent Prolog is modeled. (In fact, we view CP as a kind of subset of $\mathscr{L}$.) First note that the interpretation function f depends on atomic actions and states; for one state an atomic action can terminate and for another state it can deadlock. In the CP translation a state is a substitution. As the computation proceeds, substitutions become more filled in. Hence it is possible that at a certain point an atomic action is in a deadlock situation and at a later point (when other processes have presented more information to the state) it can continue its execution (cf. also the definition of $mgu_?$ in Appendix B for this phenomenon). One could say that a deadlock is not always hard in the sense that if a deadlock is encountered, it will not necessarily stay in that situation. We thus see that the synchronization is via shared variables, not via explicit communication actions.

This translation induces an operational- and denotational semantics for Concurrent Prolog. We combine the translation to $\mathscr{L}$ with the operational- and denotational semantics for $\mathscr{L}$. Also the equivalence (an operator linking the two semantics for Concurrent Prolog) is induced by the translation; we already have the restriction operator *restr* that related the two semantics for $\mathscr{L}$. This method of uniform abstraction gives in our opinion more insight into the semantic models than a direct definition would give. A direct definition yields a transition system in the style of [33] and a denotational semantics as in [21]. The proof of the equivalence between

two such semantic definitions would be more difficult to understand (due to the interpretation of the abstract sets).

Note that the renaming of variables takes place at the level of syntax. An alternative would be to treat this renaming at a semantic level (for example in the states).

Acknowledgment

We acknowledge fruitful discussions on our work in the Amsterdam concurrency group, including Frank de Boer, Arie de Bruin, John-Jules Meijer, Jan Rutten and Erik de Vink. We thank Erik de Vink and Katiuscia Palamidessi for the comments and suggestions made during the work. We also acknowledge useful discussions with E. Shapiro and M. Murakami during the FGCS conference on deadlock in concurrent logic languages.

Appendix A: Well-definedness of Ψ

In this appendix we show that the function Ψ as defined in Definition 5.5 is well-defined; we show that $\Psi(F)(s)(\sigma)$ is a compact set for any F, s, σ. Before we state this as a lemma, we first define another transition system which enables us to analyse a computation $(s, \sigma) \xrightarrow{z} (s', \sigma')$ by giving intermediate statements and states. When we have established relations between this new transition system to the old one, we are able to use a standard method to show compactness.

We first extend the language $\mathcal{L}$ to $\mathcal{L}^{\text{ext}}$ by introducing the operators leftmerge $\lfloor\!\lfloor$ and rightmerge $\rfloor\!\rfloor$:

$$s ::= b \mid P \mid s_1; s_2 \mid s_1 + s_2 \mid s_1 \| s_2 \mid [s] \mid s_1 \lfloor\!\lfloor s_2 \mid s_1 \rfloor\!\rfloor s_2.$$

The intuition behind the $\lfloor\!\lfloor$ ($\rfloor\!\rfloor$) is that $s_1 \lfloor\!\lfloor s_2$ ($s_1 \rfloor\!\rfloor s_2$) is like $s_1 \| s_2$, but the first step *has* to be taken from s_1 (s_2). Next we give the transition relation

$$\rightarrow_3 \subset \mathcal{L}^{\text{ext}} \times \Sigma \times (\mathcal{L}^{\text{ext}} \cup \{E\}) \times (\Sigma \cup \{\delta\}).$$

It is the union of transition relations

$$\rightarrow_1, \rightarrow_2 \subset \mathcal{L}^{\text{ext}} \times \Sigma \times (\mathcal{L}^{\text{ext}} \cup \{E\}) \times (\Sigma \cup \{\delta\}).$$

The intuition behind $\rightarrow_1$ and $\rightarrow_2$ is that we do $\rightarrow_2$ transitions inside atomic brackets and that we do $\rightarrow_1$ transitions as long as we are outside the scope of such brackets. The transition relations $\rightarrow_1, \rightarrow_2$ are defined as follows.

Definition A.1. Let

$$\rightarrow_1, \rightarrow_2 \subset \mathcal{L}^{\text{ext}} \times \Sigma \times (\mathcal{L}^{\text{ext}} \cup \{E\}) \times (\Sigma \cup \{\delta\})$$

be the smallest relation satisfying (writing $(s, \sigma) \rightarrow_i (s', \sigma')$ for $(s, \sigma, s', \sigma') \in \rightarrow_i$ and $(s, \sigma) \rightarrow_i (E, \sigma')$ for $(s, \sigma, E, \sigma') \in \rightarrow_i$ and $(s, \sigma) \rightarrow_i (E, \delta)$ for $(s, \sigma, E, \delta) \in \rightarrow_i$ $(i = 1, 2)$

- $(b, \sigma) \rightarrow_1 (E, f(b)(\sigma))$ if $f(b)(\sigma)$ exists,
- $(b, \sigma) \rightarrow_1 (E, \delta)$ if $f(b)(\sigma)$ is undefined,
- $(d(P), \sigma) \rightarrow_i (s, \sigma') \Rightarrow (P, \sigma) \rightarrow_i (s, \sigma')$ $(i = 1, 2)$,
- $(d(P), \sigma) \rightarrow_i (E, \sigma') \Rightarrow (P, \sigma) \rightarrow_i (E, \sigma')$ $(i = 1, 2)$,
- $(d(P), \sigma) \rightarrow_i (E, \delta) \Rightarrow (P, \sigma) \rightarrow_i (E, \delta)$ $(i = 1, 2)$,

- $(s_1, \sigma_1) \rightarrow_i (s_2, \sigma_2) \Rightarrow (s_1; s, \sigma_1) \rightarrow_1 (s_2; s, \sigma_2)$
  $(s_1 + s, \sigma_1) \rightarrow_1 (s_2, \sigma_2) | (E, \sigma_2) | (E, \delta)$
  $(s + s_1, \sigma_1) \rightarrow_1 (s_2, \sigma_2)$
  $(s \parallel s_1, \sigma_1) \rightarrow_1 (s \parallel s_2, \sigma_2)$
  $(s_1 \parallel s, \sigma_1) \rightarrow_1 (s_2 \parallel s, \sigma_2)$
  $(s_1 \mathbin{\lfloor\!\lfloor} s, \sigma_1) \rightarrow_1 (s_2 \parallel s, \sigma_2)$
  $(s \mathbin{\rfloor\!\rfloor} s_1, \sigma_1) \rightarrow_1 (s \parallel s_2, \sigma_2)$
  $([s_1], \sigma_1) \rightarrow_2 ([s_2], \sigma_2)$,

- $(s_1, \sigma_1) \rightarrow_1 (E, \sigma_2) \Rightarrow (s_1; s, \sigma_1) \rightarrow_1 (s, \sigma_2)$
  $(s_1 + s, \sigma_1) \rightarrow_1 (E, \sigma_2)$
  $(s + s_1, \sigma_1) \rightarrow_1 (E, \sigma_2)$
  $(s \parallel s_1, \sigma_1) \rightarrow_1 (s, \sigma_2)$
  $(s_1 \parallel s, \sigma_1) \rightarrow_1 (s, \sigma_2)$
  $(s_1 \mathbin{\lfloor\!\lfloor} s, \sigma_1) \rightarrow_1 (s, \sigma_2)$
  $(s \mathbin{\rfloor\!\rfloor} s_1, \sigma_1) \rightarrow_1 (s, \sigma_2)$
  $([s_1], \sigma_1) \rightarrow_2 (E, \sigma_2)$,

- $(s_1, \sigma_1) \rightarrow_1 (E, \delta) \Rightarrow (s_1; s, \sigma_1) \rightarrow_1 (E, \delta)$
  $(s_1 + s, \sigma_1) \rightarrow_1 (E, \delta)$
  $(s + s_1, \sigma_1) \rightarrow_1 (E, \delta)$
  $(s \parallel s_1, \sigma_1) \rightarrow_1 (E, \delta)$
  $(s_1 \parallel s, \sigma_1) \rightarrow_1 (E, \delta)$
  $(s_1 \mathbin{\lfloor\!\lfloor} s, \sigma_1) \rightarrow_1 (E, \delta)$
  $(s \mathbin{\rfloor\!\rfloor} s_1, \sigma_1) \rightarrow_1 (E, \delta)$
  $([s_1], \sigma_1) \rightarrow_2 (E, \delta)$,

- $(s_1, \sigma_1) \rightarrow_2 (s_2, \sigma_2) \Rightarrow (s_1; s, \sigma_1) \rightarrow_2 (s_2; s, \sigma_2)$
  $(s_1 + s, \sigma_1) \rightarrow_2 (s_2, \sigma_2)$
  $(s + s_1, \sigma_1) \rightarrow_2 (s_2, \sigma_2)$
  $(s \parallel s, \sigma_1) \rightarrow_2 (s_2 \mathbin{\lfloor\!\lfloor} s, \sigma_2)$
  $(s \parallel s_1, \sigma_1) \rightarrow_2 (s \mathbin{\rfloor\!\rfloor} s_2, \sigma_2)$
  $(s_1 \mathbin{\lfloor\!\lfloor} s, \sigma_1) \rightarrow_2 (s_2 \mathbin{\lfloor\!\lfloor} s, \sigma_2)$
  $(s \mathbin{\rfloor\!\rfloor} s_1, \sigma_1) \rightarrow_2 (s \mathbin{\rfloor\!\rfloor} s_2, \sigma_2)$
  $([s_1], \sigma_1) \rightarrow_2 ([s_2], \sigma_2)$,

- $(s_1, \sigma_1) \to_2 (E, \sigma_2) \Rightarrow (s_1; s, \sigma_1) \to_2 (s, \sigma_2)$
  $(s_1 + s, \sigma_1) \to_2 (E, \sigma_2)$
  $(a + s_1, \sigma_1) \to_2 (E, \sigma_2)$
  $(s_1 \| s, \sigma_1) \to_2 (s, \sigma_2)$
  $(s \| s_1, \sigma_1) \to_2 (s, \sigma_2)$
  $(s_1 \lfloor\!\lfloor s, \sigma_1) \to_2 (s, \sigma_2)$
  $(s \lfloor\!\lfloor s_1, \sigma_1) \to_2 (s, \sigma_2)$
  $([s_1], \sigma_1) \to_2 (E, \sigma_2)$,

- $(s_1, \sigma_1) \to_2 (E, \delta) \Rightarrow (s_1; s, \sigma_1) \to_2 (E, \delta)$
  $(s_1 + s, \sigma_1) \to_2 (E, \delta)$
  $(s + s_1, \sigma_1) \to_2 (E, \delta)$
  $(s_1 \| s, \sigma_1) \to_2 (E, \delta)$
  $(s \| s_1, \sigma_1) \to_2 (E, \delta)$
  $(s_1 \lfloor\!\lfloor s, \sigma_1) \to_2 (E, \delta)$
  $(s \lfloor\!\lfloor s_1, \sigma_1) \to_2 (E, \delta)$
  $([s_1], \sigma_1) \to_2 (E, \delta)$.

Note that transitions of the form $(s_1 \text{ op } s_2, \sigma) \to_1 \cdots$ for $\text{op} \in \{\lfloor\!\lfloor, \lfloor\!\lfloor\}$ are not needed if we consider only transition sequences from (s, σ) where $s \in \mathcal{L}$.

Lemma A.2. *For any* $(s, \sigma) \in \mathcal{L}^{\text{ext}} \times \Sigma$ *the sets*

$$\{(s', \sigma') : (s, \sigma) \to_3 (s', \sigma')\}$$

and

$$\{(E, \sigma') : (s, \sigma) \to_3 (E, \sigma')\}$$

are finite.

Next we give a relationship between the transition relations.

Lemma 7.3. *For all* $n \geq 0$, s, s', σ, σ', $\sigma_1, \ldots, \sigma_n, \ldots$ *we have*

(1) $(s, \sigma) \xrightarrow{\sigma_1 \ldots \sigma_n \sigma'} (E, \sigma') \Rightarrow \exists s_1, \ldots, s_n$
$\in \mathcal{L}^{\text{ext}}$ $[(s, \sigma) \to_3 (s_1, \sigma_1) \to_3 \cdots (s_n, \sigma_n) \to_3 (E, \sigma')]$,

(2) $(s, \sigma) \xrightarrow{\sigma_1 \ldots \sigma_n \sigma'} (s', \sigma') \Rightarrow \exists s_1, \ldots, s_n$
$\in \mathcal{L}^{\text{ext}}$ $[(s, \sigma) \to_3 (s_1, \sigma_1) \to_3 \cdots (s_n, \sigma_n) \to_3 (s', \sigma')]$,

(3) $(s, \sigma) \xrightarrow{\alpha_1 \ldots \sigma_n \delta} (E, \delta) \Rightarrow \exists s_1, \ldots, s_n$
$\in \mathcal{L}^{\text{ext}}$ $[(s, \sigma) \to_3 (s_1, \sigma_1) \to_3 \cdots (s_n, \sigma_n) \to_3 (E, \delta)]$,

(4) $(s, \sigma) \xrightarrow{\sigma_1 \sigma_2 \cdots} (E, \delta) \Rightarrow \exists s_1, s_2, \ldots$
$\in \mathcal{L}^{\text{ext}}$ $[(s, \sigma) \to_3 (s_1, \sigma_1) \to_3 (s_2, \sigma_2) \to_3 \cdots]$.

The inverse implications are *not* true in general; take b_1, b_2 such that $f(b_1)(\sigma) = \sigma$ and $f(b_2)(\sigma)$ is undefined for all σ. We have

(1) $(b_1; b_1, \sigma) \to_3 (b_1, \sigma) \to_3 (E, \sigma)$ but not $(b_1; b_1, \sigma) \xrightarrow{\sigma\sigma} (E, \sigma)$,

(2) $(b_1; b_1; b_1, \sigma) \to_3 (b_1; b_1, \sigma) \to_3 (b_1, \sigma)$ but not $(b_1; b_1; b_1, \sigma) \xrightarrow{\sigma\sigma} (b_1, \sigma)$

(3) $(b_1; b_2, \sigma) \to_3 (b_2, \sigma) \to_3 (E, \delta)$ but not $(b_1; b_2, \sigma) \xrightarrow{\sigma\sigma} (E, \delta)$,

(4) $d(P) = b_1$, P and $(P, \sigma) \to_3 (P, \sigma) \to_3 (P, \sigma) \to_3 \cdots$ but not $(P, \sigma) \xrightarrow{\sigma\sigma\cdots} (E, \delta)$.

We can summarize these counterexamples above by saying that we do not have enough information about the grain size in the sequence of $\to_3$ transitions.

We introduce the notion of a substatement. A substatement intuitively is a part of a statement that can perform one step. The function *sub* delivers a set of substatements.

Definition 7.4

(1) $sub(b) = \{b\}$,

(2) $sub(s_1; s_2) = sub(s_1)$,

(3) $sub(s_1 + s_2) = sub(s_1) \cup sub(s_2)$,

(4) $sub(s_1 \| s_2) = sub(s_1) \cup sub(s_2)$,

(5) $sub([s_1]) = \{[s_1]\}$,

(6) $sub(P) = sub(g)$ if $d(P) = g$.

With the notion of substatement we are able to state the following lemma.

Lemma 7.5. *For all $n \geq 0$, s, σ, σ', $\sigma_1, \ldots, \sigma_n, \ldots$ we have*

(1) $\exists s' \quad [(s, \sigma) \xrightarrow{\sigma_1 \cdots \sigma_n \sigma'} (s', \sigma')] \vee (s, \sigma) \xrightarrow{\sigma_1 \cdots \sigma_n \sigma'} (E, \sigma') \Leftrightarrow$
$\exists \tilde{s} \in sub(s) \exists s_1, \ldots, s_n$
$\in \mathscr{L}^{\text{ext}} \quad [(\tilde{s}, \sigma) \to_3 (s_1, \sigma_1) \to_3 \cdots (s_n, \sigma_n) \to_3 (E, \sigma')]$,

(2) $(s, \sigma) \xrightarrow{\sigma_1 \cdots \sigma_n \delta} (E, \delta) \Leftrightarrow \exists \tilde{s} \in sub(s) \exists s_1, \ldots, s_n$
$\in \mathscr{L}^{\text{ext}} \quad [(\tilde{s}, \sigma) \to_3 (s_1, \sigma_1) \to_3 \cdots (s_n, \sigma_n) \to_3 (E, \delta)]$,

(3) $(s, \sigma) \xrightarrow{\sigma_1 \sigma_2 \cdots} (E, \delta) \Leftrightarrow$
$\exists \tilde{s} \in sub(s) \exists s_1, s_2, \ldots \in \mathscr{L}^{\text{ext}} \quad [(\tilde{s}, \sigma) \to_3 (s_1, \sigma_1) \to_3 (s_2, \sigma_2) \to_3 \cdots]$.

Now we turn to the well-definedness of Ψ.

Lemma 7.6. *We have that $\Psi(F)(s)(\sigma)$ is a compact set for any F, s, σ.*

Proof. Pick arbitrary F, s, σ. Take an arbitrary infinite sequence in $\Psi(F)(s)(\sigma)$. There exists an infinite subsequence of this subsequence in one of the following three subsets of $\Psi(F)(s)(\sigma)$:

(1) $\{z : (s, \sigma) \xrightarrow{z} (E, \sigma')\}$,

(2) $\{z : (s, \sigma) \xrightarrow{z} (E, \delta)\}$,

(3) $\{(z, F(s')) : (s, \sigma) \xrightarrow{z} (s', \sigma')\}$.

We only consider the last case. The other cases can be handled in a similar way. First note that there is only a finite number of statements s' such that

$$\exists\sigma'\exists z \quad [(s,\sigma) \xrightarrow{z} (s',\sigma')].$$

Hence there exists an infinite subsequence of the form $(z_i, F(s'))_i$ for some fixed s'. Hence it suffices to show that $(z_i)_i$ has a converging subsequence. Assume there is no infinite constant subsequence (otherwise we are done).

For all i there exists a statement s_i' and σ_i such that $(s,\sigma) \to^{z_i} (s_i', \sigma_i')$. By Lemma 7.5 we have that for each i there exist a substatement $\tilde{s}_i \in sub(s)$, integer $n_i \geq 0$, tuples $(s_{i1}, \sigma_{i1}), \ldots, (s_{in_i}, \sigma_{in_i})$ such that

$$z_i = \sigma_{i1}\ldots\sigma_{in_i}\sigma_i', \qquad (\tilde{s}_i,\sigma) \to_3 (s_{i1},\sigma_{i1}) \to_3 \cdots (s_{in_i},\sigma_{in_i}) \to_3 (E,\sigma_i').$$

There exists only a finite number of substatements of s. Hence an infinite number of the $\tilde{s}_i$ is equal to a certain $\tilde{s}$. So we are able to pick an infinite subsequence $(z_{f(i)})_i$ of $(z_i)_i$ where f is chosen such that f is monotonic and $\tilde{s}_{f(i)} = \tilde{s}$ for all i. For each $n \geq 1$ we can pick a tuple (s_n, σ_n) and a monotonic function $f_n : \mathbb{N} \to \mathbb{N}$ such that

(1) if $n = 1$ then $(z_{(f_1 \circ f)(i)})_i$ is an infinite subsequence of $(z_{f(i)})_i$,

(2) if $n > 1$ then $(z_{(f_{n+1}\circ\cdots\circ f_1\circ f)(i)})_i$ is an infinite subsequence of $(z_{(f_n\circ\cdots\circ f_1\circ f)(i)})_i$,

(3) $\forall i\,[(s_{(f_n\circ\cdots\circ f_1\circ f)(i),n}, \sigma_{(f_n\circ\cdots\circ f_1\circ f)(i),n}) = (s_n,\sigma_n)]$.

We have

$$(\tilde{s},\sigma) \to_3 (s_1,\sigma_1) \to_3 (s_2,\sigma_2) \to_3 \cdots.$$

Take the infinite subsequence $(z_{(f_i\circ\cdots\circ f_1)(i)})_i$ of $(z_i)_i$. Note that

$$\lim_{i\to\infty} z_{(f_i\circ\cdots\circ f_1\circ f)(i)} = \sigma_1\sigma_2\cdots =^{\text{def}} z$$

and by Lemma 7.5, $(s,\sigma) \to^z (E,\delta)$.

Appendix B: The extended unification function

In this appendix we show a way to define the function $mgu_?$ based on [33]. (Following [33] we interpret the read-only functor ? as an input-only functor.) There are other ways to define this function. This paper is independent of the way the $mgu_?$ function is defined.

Following Saraswat, we impose the following restrictions:

(1) read-only functors are only allowed in heads of clauses,

(2) if a subterm of a term is annotated, then also the term is annotated.

We formalize these points. Let t be a typical element of the set *Term* of terms in which the read-only functor ? does not appear:

$$t ::= x \mid f(t_1,\ldots,t_n) \quad (f \neq ? \wedge n \geq 0).$$

We extend *Term* to $Term_?$ as follows. Let s be a typical element of the set $Term_?$:

$$s ::= t \mid ?(x) \mid ?(f(s_1, \ldots, s_n)) \quad (f \neq ? \wedge n \geq 0).$$

Let *Atom* contain elements of the form $P(t_1, \ldots, t_n)$ and $Atom_?$ elements of the form $P(s_1, \ldots, s_n)$ where P is a predicate symbol of arity n. The restrictions can now be stated as follows; elements of $Atom_?$ are only allowed as the heads of clauses. All other atoms (in goals, guards and bodies) are to be taken from *Atom*.

Assume that a variable in the head of a clause is annotated (preceded by a read-only functor). The meaning of this annotation is that the variable should receive a (partial) value (i.e. a term that is not a variable) when it is unified with an atom in the goal.

We give an extended version of the unification algorithm based on [2] and [11]. A finite subset of $Term_? \times Term_?$ we call *solved* if it is of the form

$$\{\langle x_1, t_1\rangle, \ldots, \langle x_n, t_n\rangle\}$$

where for $1 \leq i \leq n$ we have that $x_i \in Var$, $t_i \in Term$, x_i does not occur in t_i and for $1 \leq i < j \leq n$ we have $x_i \neq x_j$. A finite solved subset of $Term_? \times Term_?$ determines a substitution θ as follows:

$$\theta(x_i) = t_i \quad (1 \leq i \leq n), \qquad \theta(x) = x \quad (x \notin \{x_1, \ldots, x_n\}).$$

Assume that $X, Y \subset Term_? \times Term_?$. We write $X \to Y$ if Y is obtained from X by choosing an element of one of the forms below and by performing the corresponding action.

(1) $\langle x, x\rangle$: delete the pair,
(2) $\langle ?(x), f(t_1, \ldots, t_n)\rangle$: replace by $\langle x, f(t_1, \ldots, t_n)\rangle$,
(3) $\langle f(t_1, \ldots, t_n), ?(x)\rangle$: replace by $\langle f(t_1, \ldots, t_n), x\rangle$,
(4) $\langle ?(f(s_1, \ldots, s_n)), f(t_1, \ldots, t_n)\rangle$: replace by $\langle s_1, t_1\rangle, \ldots, \langle s_n, t_n\rangle$,
(5) $\langle f(t_1, \ldots, t_n), ?(f(s_1, \ldots, s_n))\rangle$: replace by $\langle t_1, s_1\rangle, \ldots, \langle t_n, s_n\rangle$,
(6) $\langle f(t_1, \ldots, t_n), f(t'_1, \ldots, t'_n))\rangle$: replace by $\langle t_1, t'_1\rangle, \ldots, \langle t_n, t'_n\rangle$,
(7) $\langle t, x\rangle$: replace by $\langle x, t\rangle$,
(8) $\langle x, t\rangle$ where x occurs in other pairs and x does not occur in t: apply substitution $x := t$ to all other pairs.

Now define

$$mgu_?: Atom_? \times Atom_? \to Subst$$

by

$$mgu_?(P(s_1, \ldots, s_n), P(s'_1, \ldots, s'_n)) = \theta$$

if there exist a $n \geq 0$ and $X_1, X_2, \ldots, X_n$ such that

$$\{\langle s_1, s'_1\rangle, \ldots, \langle s_n, s'_n\rangle\} \to X_1 \to X_2 \to \cdots \to X_n$$

where X_n is in solved form and determines θ, and is undefined otherwise.

References

[1] P. America and J.J.M.M. Rutten, Solving reflexive domain equations in a category of complete metric spaces, *J. Comput. System Sci.* **39**(3) (1989) 343-375.

[2] K.R. Apt, Introduction to logic programming, Technical Report CS-R8741, Centre for Mathematics and Computer Science, Amsterdam, 1987. To appear in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science* (North-Holland, Amsterdam).

[3] K.R. Apt and G. Plotkin, Countable nondeterminism and random assignment, *J. ACM* **33**(4) (1986) 724-767.

[4] K.R. Apt and M.H. van Emden, Contributions to the theory of logic programming. *J. ACM* **29**(3) (1982) 841-862.

[5] B. Arbab and D.M. Berry, Operational and denotational semantics of prolog, *J. Logic Programming* **4** (1987) 309-330.

[6] L. Beckman, Towards a formal semantics for concurrent logic programming languages, in: E. Shapiro, ed., *Proc. Third Internat. Conf. on Logic Programming*, Lecture Notes in Computer Science, **225** (Springer, Berlin, 1986) 335-349.

[7] K.L. Clark and S. Gregory, Parallel programming in logic, *ACM Trans. Programming Language Systems* **8**(1) (1986) 1-49.

[8] J.W. de Bakker, Comparative semantics for flow of control in logic programming without logic, Technical Report CS-R8840, Centre for Mathematics and Computer Science, Amsterdam, 1988.

[9] J.W. de Bakker and J.-J.Ch. Meyer, Metric semantics for concurrency. *BIT* **28** (1988) 504-529.

[10] J.W. de Bakker and J.I. Zucker Processes and the denotational semantics of concurrency, *Inform. and Control* **54** (1982) 70-120.

[11] F.S. de Boer, J.N. Kok, C. Palamidessi and J.J.M.M. Rutten, Control flow versus logic: a denotational and a declarative model for guarded horn clauses, in: A. Kreczmar and G. Mirkowska, eds., *Proc. Mathematical Foundations of Computer Science* (*MFCS 89*), Lecture Notes in Computer Science **379** (Springer, Berlin, 1989) 165-176..

[12] F.S. de Boer, J.N. Kok, C. Palamidessi and J.J.M.M. Rutten, Semantic models for a version of parlog, in: G. Levi and M. Martelli, eds., *Proc. Internat. Conf. on Logic Programming* (*ICLP 89*) (MIT Press, Cambridge, MA, 1989) 621-636.

[13] A. de Bruin and E. de Vink, Continuation semantics for prolog with cut, in: J. Diax and F. Orejas, eds., *Proc. CAAP 89*, Lecture Notes in Computer Science **351** (Springer, Berlin, 1989).

[14] S.K. Debray and P. Mishra, Denotational and operational semantics for prolog, in: M. Wirsing, ed., *Formal Description of Programming Concepts III* (North-Holland, Amsterdam, 1987) 245-269.

[15] R. Engelking, *General Topology* (Polish Scientific Publishers, 1977).

[16] M. Falaschi and G. Levi, Finite failures and partial computations in concurrent logic languages, in: *Proc. Fifth Generation Computer Systems* (*FGCS 88*) (1988) 364-373.

[17] R. Gerth, M. Codish, Y. Lichtenstein and E. Shapiro, Fully abstract denotational semantics for concurrent prolog, in: *Proc. Logic in Computer Science* (*LICS 88*) (1988) 320-335.

[18] M. Hennessy and G.D. Plotkin, Full abstraction for a simple parallel programming language, in: J. Becvar, ed., *Proc. Mathematical Foundations of Computer Science* (*MFCS 79*), Lecture Notes in Computer Science **74** (Springer, Berlin, 1979) 108-120.

[19] N.D. Jones and A. Mycroft, Stepwise development of operational and denotational semantics for prolog, in: *Proc. Internat. Symp. on Logic Programming* (*ICLP 84*) (IEEE, 1984) 281-288.

[20] P. Knijnenburg and J.N. Kok, A compositional semantics for the finite and infinite failures of a language with atomized statements, Technical report, University of Utrecht, 1989. To appear in: *Proc. CSN 89.*

[21] J.N. Kok, A compositional semantics for concurrent prolog, in: R. Cori and M. Wirsing, eds., *Proc. Symp. on Theoretical Aspects Computer Science* (*STACS 88*), Lecture Notes in Computer Science **294** (Springer, Berlin, 1988) 373-388.

[22] J.N. Kok and J.J.M.M. Rutten, Contractions in comparing concurrency semantics, in: T. Lepistö and A. Salomaa, eds., *Proc. Internat. Colloquium Automata, Languages and Programming* (*ICALP 88*), Lecture Notes in Computer Science **317** (Springer, Berlin, 1988) 317-332. To appear in: *Theoret. Comput. Sci.*

[23] G. Levi, A new declarative semantics of flat guarded horn clauses, Technical report, ICOT, Tokyo, 1988.

[24] G. Levi and C. Palamidessi, The declarative semantics of logical read-only variables, in: *Proc. Symp. on Logic Programming (SLP 85)* (IEEE Comp. Society Press, 1985) 128-137.

[25] G. Levi and C. Palamidessi, An approach to the declarative semantics of synchronization in logic languages, in: *Proc. Internat. Conf. on Logic Programming (ICLP 87)* (1987) 877-893.

[26] J.W. Lloyd, *Foundations of Logic Programming*, 2nd edn. (Springer, Berlin, 1987).

[27] C. Mierkowsky, S. Taylor, E. Shapiro, J. Levy and M. Safra, The design and implementation of flat concurrent prolog, Technical Report CS85-09, Weizmann Institute, Dept. of Applied Maths, Israel, 1985.

[28] M. Murakami, A declarative semantics of parallel logic programs with perpertual processes, in: *Proc. Fifth Generation Computer Systems (FGCS 88)* (1988) 374-381.

[29] E.-R. Olderog and C.A.R. Hoare, Specification-oriented semantics for communicating processes, *Acta Inform.* **23** (1986) 9-66.

[30] G.D. Plotkin, A structural approach to operational semantics, Technical Report DAIMI FN-19, Aarhus Univ., Comp. Sci. Dept., 1981.

[31] G.A. Ringwood, Parlog 86 and the dining logicians, *Comm. ACM* **31** (1988) 10-25.

[32] J.J.M.M. Rutten, Correctness and full abstraction of metric semantics for concurrency, in: J.W. de Bakker, W.P. de Roever and G. Rozenberg, eds., *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, Lecture Notes in Computer Science **354** (Springer, Berlin, 1988) 628-659.

[33] V.A. Saraswat, The concurrent logic programming language cp: definition and operational semantics, in: *Conf. Record of the Fourteenth Ann. ACM Symp. on Principles of Programming Languages* (1987) 49-62.

[34] V.A. Saraswat, Concurrent Constraint Programming Languages, PhD thesis, Carnegie-Mellon University, 1989.

[35] E.Y. Shapiro, A subset of concurrent prolog and its interpreter, Technical Report TR-003, ICOT, Tokyo, 1983.

[36] E.Y. Shapiro, Concurrent prolog, a progress report, in: W. Bibel and Ph. Jorrand, eds., *Fundamentals of Artificial Intelligence*, Lecture Notes in Computer Science **232** (Springer, Berlin, 1987).

[37] E.Y. Shapiro, *Concurrent Prolog: Collected Papers*, Vols. 1, 2 (MIT Press, Cambridge, MA, 1988).

[38] K. Ueda, Guarded horn clauses, Technical Report TR-103, ICOT, Tokyo, 1985. Revised in 1986. A revised version is in: E. Wada, ed., *Proc. Logic Programming '85*, Lecture Notes in Computer Science **221** (Springer, Berlin, 1989) 168-179. Also in: E.Y. Shapiro, ed., *Concurrent Prolog: Collected Papers* (MIT Press, Cambridge, MA, 1988) Chap. 4.

[39] K. Ueda, Guarded horn clauses: a parallel logic programming language with the concept of a guard, Technical Report TR-208, ICOT, Tokyo, 1986. Revised in 1987. Also in M. Nivat and K. Fuchi, eds., *Proc. Programming of Future Generation Computers* (North-Holland, Amsterdam, 1988) 441-456.